Version 1.0 - Oct. 18, 2000

Writing ImageJ PlugIns – A Tutorial

Werner Bailer

Werner.Bailer@fhs-hagenberg.ac.at

Fachhochschule Hagenberg, Austria
Medientechnik und -design

1 Getting Started

1.1 About ImageJ[1]

ImageJ is a public domain Java image processing program inspired by NIH Image for the Macintosh. It runs, either as an online applet or as a downloadable application, on any computer with a Java 1.1 or later virtual machine.

It can display, edit, analyze, process, save and print 8-bit, 16-bit and 32-bit images. It can read many image formats including TIFF, GIF, JPEG, BMP, DICOM, FITS and "raw". It supports "stacks", a series of images that share a single window.

It is multithreaded, so time-consuming operations such as image file reading can be performed in parallel with other operations.

It can calculate area and pixel value statistics of user-defined selections. It can measure distances and angles. It can create density histograms and line profile plots. It supports standard image processing functions such as contrast manipulation, sharpening, smoothing, edge detection and median filtering.

It does geometric transformations such as scaling, rotation and flips. Image can be zoomed up to 32:1 and down to 1:32. All analysis and processing functions are available at any magnification factor. The program supports any number of windows (images) simultaneously, limited only by available memory.

Spatial calibration is available to provide real world dimensional measurements in units such as millimeters. Density or gray scale calibration is also available.

ImageJ was designed with an open architecture that provides extensibility via Java plugins. Custom acquisition, analysis and processing plugins can be developed using ImageJ's built in editor and Java compiler. User-written plugins make it possible to solve almost any image processing or analysis problem.

ImageJ is being developed using Metrowerks CodeWarrior, and the source code is freely available. The author, Wayne Rasband (wayne@codon.nih.gov), is at the Research Services Branch, National Institute of Mental Health, Bethesda, Maryland, USA.

1.2 About this Tutorial

This tutorial is an introduction to writin plugins for ImageJ. It explains the concept of plugins in ImageJ, starting with the sample plugins that are part of the ImageJ distribution and covers the parts of those Im-

[1] Description taken from http://rsb.info.nih.gov/ij/docs/intro.html

ageJ API, that are essential for writing plugins. A reference of the most important classes, methods and constants is provided.

A basic knowledge of the Java language is required. (Resources for Java beginners can be found in section 9.4). You should also try to get familiar with ImageJ before you start writing plugins.

For the development of plugins you need ImageJ and a Java compiler. You can write your plugins using any text editor or Java IDE (Integrated Development Environment). You can compile them using a Java compiler of your choice or inside ImageJ.

1.3 Setting up your Environment

1.3.1 Installing ImageJ

The latest distribution of ImageJ can be downloaded from

```
http://rsb.info.nih.gov/ij/download.html
```

Windows

The Windows standalone version includes a Java Runtime Environment (JRE). To install it, run the self-extracting file you have downloaded. In the destination directory you will find a link called ImageJ. Double-click this link to run ImageJ.

If you already have a JRE or JDK installed, you can use it to run ImageJ. In that case you just need to download the ImageJ JAR file.

To execute ImageJ from the command line change to the ImageJ folder and type:

```
java -cp ij.jar ij.ImageJ
```

Macintosh

To run ImageJ on a Mac you need the MacOS Runtime for Java (MRJ). It can be downloaded from http://www.apple.com/java. Installation instructions can be found on the download page.

The ImageJ distribution for the Mac just needs to be unpacked after downloading. Double-click the ImageJ icon in the newly created folder to run it.

Unix, Linux

Download the JAR file containing the ImageJ classes. A Java Runtime Environment or JDK has to be installed on your computer. If the path to `java` is set properly, change to the folder where the ImageJ JAR file is located and type

```
java -classpath ij.jar ij.ImageJ
```

1.3.2 Installing the Java Compiler

To compile plugins you need a Java compiler. It is for example included in the Java Development Kit (JDK) from Sun Microsystems for Windows, Solaris and Linux or the MacOS Runtime for Java (MRJ) Software Development Kit (SDK).

Windows

Download the JDK at http://www.javasoft.com and execute the self-installing file following the instructions on the screen. Compiling plugins and starting ImageJ using a batch file will be explained in section 3.4.

If you run ImageJ using a JDK you can compile plugins inside ImageJ. For JDK 1.2 or higher, make sure that the tools.jar file is in the classpath. Type on the command line in the ImageJ folder:

```
java -cp ij.jar;c:\jdk1.3\lib\tools.jar ij.ImageJ
```

Insert the path where the JDK is installed on your system.

Macintosh

In addition to the MRJ you need the MRJ SDK. It can be downloaded from http://developer.apple.com/java. Run the installer you have downloaded. After the installation it is possible to compile plugins inside ImageJ.

2 ImageJ Class Structure

This is an overview of the class structure of ImageJ. It is by far not complete, just the most important for plugin programming are listed and briefly described. Detailed descriptions of classes and methods can be found in the chapters 3 through 6.

`ij`

`ImageJApplet`

ImageJ can be run as applet or as application. This is the applet class of ImageJ. The advantage of running ImageJ as applet is that it can be run (remotely) inside a browser, the biggest disadvantages is the limited access to files on disk because of the Java applet security concept.

`ImageJ`

The main class of the ImageJ application. This class contains the run method which is the program's main entry point and the ImageJ main window.

`Executer`

A class for executing menu commands in separate threads (without blocking the rest of the program).

`IJ`

A class containing many utility methods (discussed in section 5).

`ImagePlus`

The representation of an image in ImageJ, which is based on an ImageProcessor (see section 4).

`ImageStack`

An ImageStack is an expandable array of images (see section 4).

`ij.gui`

`ProgressBar`

A bar in the ImageJ main window that informs graphically about the progress of a running operation.

`GenericDialog`

A modal dialog that can be customized and called on the fly, e.g. for getting user input before running a plugin (see chapter 6).

`NewImage`

A class for creating a new image of a certain type from scratch.

`Roi`

A class representing a region of interest of an image. If supported by a plugin, it can process just the ROI and not the whole image.

`ij.io`

This package contains classes for reading/decoding and writing/encoding image files.

`ij.measure`

Contains classes for measurements.

`ij.plugin`

`PlugIn`

This interface has to be implemented by plugins, that do not require an image as input (see section 3).

`ij.plugin.filter`

`PlugInFilter`

This interface has to be implemented by plugins, that require an image as input (see section 3).

`ij.plugin.frame`

`PlugInFrame`

A window class that can be subclassed by a plugin (see section 3).

`ij.process`

`ImageConverter`

A class that contains methods for converting images from one image type to another.

`ImageProcessor`

An abstract superclass of image processors for certain image types. An image processor provides methods for actually working on the image (see chapter 4).

`StackConverter`

A class for converting stacks form one image type to another.

`StackProcessor`

A class for processing image stacks.

`ij.text`

This package contains classes for displaying text.

3 The PlugIn Concept of ImageJ

The functions provided by ImageJ's menu commands (most of them are in fact plugins themselves) can be extended by user plugins. These plugins are Java classes implementing the necessary interfaces that are placed in a certain folder. They can be conveniently compiled inside ImageJ or using a batch file in a Windows environment. Plugins found by ImageJ are placed in the Plugins menu.

3.1 Types of PlugIns

There are basically two types of plugins: those that do not require an image as input (implementing the interface `PlugIn`) and plugin filters, that require an image as input (implementing the interface `PlugInFilter`).

3.2 Interfaces

PlugIn

This interface has just one method:

```
void run(java.lang.String arg)
```

This method runs the plugin, what you implement here is what the plugin actually does. `arg` is a string passed as an argument passed to the plugin, the argument may also be an empty string. You can install plugins more than once, so each of them will call the same plugin class with a different argument.

PlugInFilter

This interface also has a method

```
void run(ImageProcessor ip)
```

This method runs the plugin, what you implement here is what the plugin actually does. It takes the image processor it works on as an argument. The processor can be modified directly or a new processor and a new image can be based on its data, so that the original image is left unchanged. The original image is locked while the plugin is running. In contrast to the `PlugIn` interface the run method does not take a string argument – the argument can be passed using

```
int setup(java.lang.String arg, ImagePlus imp)
```

This method sets up the plugin filter for use. The `arg` string has the same function as in the run method of the `PlugIn` interface. You do not have to care for the argument `imp` – this is handled by ImageJ and the currently active image is passed. The `setup` method returns a flag word that represents the filter's capabilities (i.e. which types of images it can handle). The following capability flags are defined in `PlugInFilter`:

`static int DOES_16`
: The plugin filter handles 16 bit grayscale images.

`static int DOES_32`
: The plugin filter handles 32 bit floating point grayscale images.

`static int DOES_8C`
: The plugin filter handles 8 bit color images.

`static int DOES_8G`
: The plugin filter handles 8 bit grayscale images.

`static int DOES_ALL`
: The plugin filter handles all types of images.

`static int DOES_RGB`
: The plugin filter handles RGB images.

`static int DOES_STACKS`
: The plugin filter supports stacks, ImageJ will call it for each slice in a stack.

`static int DONE`
: If the `setup` method returns `DONE` the run method will not be called.

`static int NO_CHANGES`
: The plugin filter does not change the pixel data.

`static int NO_IMAGE_REQUIRED`
: The plugin filter does not require an image to be open.

`static int NO_UNDO`

The plugin filter does not require undo.

`static int ROI_REQUIRED`

The plugin filter requires a region of interest (ROI).

`Static int STACK_REQUIRED`

The plugin filter requires a stack.

`static int SUPPORTS_MASKING`

Plugin filters always work on the bounding rectangle of the ROI. If this flag is set and there is a non-rectangular ROI, ImageJ will restore the pixels that are inside the bounding rectangle but outside the ROI.

3.3 PlugIns Folder

ImageJ's user plugins have to be located in a folder called `plugins` which is a subfolder of the ImageJ folder. But only class files in the plugins folder with at least one underscore in their name appear automatically in the plugins menu.

3.4 The Compile Batch File

If you run ImageJ on a Mac or using a JDK as runtime environment on a Windows or Unix system you can compile and run plugins using the "Compile and run" command in the Plugins menu.

To compile from the command line in a Windows environment you have to edit the file `compile.bat` file that is located in the `plugins` folder. The file looks like this:

```
rem Compiles all the plugins in this folder and then runs ImageJ
set PATH=c:\jdk1.3\bin;%PATH%
set CLASSPATH=../ij.jar;.
javac *.java
java ij.ImageJ
```

In the second line, insert the path where the JDK is installed on your system.

3.5 A Sample PlugIn (Example)

If you look into the `plugins` folder right after installing ImageJ you will find the sample plugins that come with ImageJ. In this section we will take a closer look at one of them.

`Inverter_` is a plugin that inverts 8 bit grayscale images.

Here we import the necessary packages, `ij.*` for the basic ImageJ classes, `ij.process.*` for image processors and the interface `ij.plugin.filter.PlugInFilter` is the interface we have to implement for a plugin filter.

```
import ij.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;
import java.awt.*;
```

NOTE: Do not use a `package` statement inside plugin classes – they have to be in the default package!

Our plugin has the necessary underscore appended. It needs an image as input, it has to implement `PlugInFilter`:

```
public class Inverter_ implements PlugInFilter {
  ...
```

What comes next is the method for setting up the plugin. For the case that we get "about" as argument, we call the method `showAbout` that displays an about dialog. In that case we return `DONE` because we do not want the `run` method to be called. In any other case we return the capability flags for this plugin: It works on 8 bit grayscale images, also on stacks and in the case that there is a ROI defined the plugin will just work on the masked region (region of interest, ROI).

```
public int setup(String arg, ImagePlus imp) {
   if (arg.equals("about"))
      {showAbout(); return DONE;}
   return DOES_8G+DOES_STACKS+SUPPORTS_MASKING;
}
```

The `run` method implements the actual function of the plugin. We get the processor of the original image. Then we get the image as an array of pixels from the processor – as it is a 8 bit grayscale image (= 256 possible values) we can use a `byte` array. Note that the image is one-dimensional, containing one scan line after the other. Then we read the width of the image (because we need to know the length of a scan line) and the bounding rectangle of the ROI.

```
public void run(ImageProcessor ip) {
   byte[] pixels = (byte[])ip.getPixels();
   int width = ip.getWidth();
   Rectangle r = ip.getRoi();
```

We now declare two variables to avoid calculating the position in the one dimensional image array every time. In the outer loop we go from the first line of the ROI to its last line. We calculate the offset (= position of the first pixel of the current scan line) and go in the inner loop from the left most pixel of the ROI to its right most pixel. We assign the current position to `i` and invert the pixel value by subtracting it's value from 255.

```
int offset, i;
for (int y=r.y; y<(r.y+r.height); y++) {
   offset = y*width;
   for (int x=r.x; x<(r.x+r.width); x++) {
   i = offset + x;
      pixels[i] = (byte)(255-pixels[i]);
   }
}
```

`showAbout` uses the static method `showMessage` from class `IJ` to display a text in a message box. The first parameter specifies its title, the second the message text.

```
void showAbout() {
   IJ.showMessage("About Inverter_...",
   "This sample plugin filter inverts 8-bit images. Look\n" +
   "at the 'Inverter_.java' source file to see how easy it is\n" +
   "in ImageJ to process non-rectangular ROIs, to process\n" +
   "all the slices in a stack, and to display an About box."
   );
}
```

3.6 Compiling and Running PlugIns

Now that we have looked at one of the sample plugins we want to compile and run it.

In a Windows environment it is possible to call the batch file `compile.bat` in the `plugins` folder. This will compile all Java source files in this folder and run ImageJ.

The compiled plugins (those with at least one underscore in their name) will become available in the Plugins menu.

On a Mac or if the Java runtime environment you use includes a Java compiler (javac) you can compile and run plugins by selecting Compile and run... from the Plugins menu.

3.7 Integrating PlugIns into the ImageJ GUI

Like commands plugins can be accessed via hot-keys. You can create a new hot-key by selecting "Create hot-key" from the menu "Plugins / Hot-Keys".

When we discussed the plugin interfaces we talked about arguments that can be passed to plugins. Installing a plugin using the menu command Plugins / Hot-Keys / Install Plugin ... places the plugin into a selected menu, assigns a hot-key and passes an argument.

Plugins / Hot-Keys / Remove ... removes a plugin from the menu.

4 Image Representation in ImageJ

When we looked at the sample plugin in the previous section we saw that images are represented by ImagePlus and ImageProcessor objects in ImageJ. In this section we take a closer look at the way images are handled by ImageJ. Methods that are not discussed in the text but are of some importance for writing plugins can be found in the reference in section 4.11.

4.1 Types of Images

Images are large arrays of pixel values. But it is important to know how these pixel values should be interpreted. This is specified by the type of the image. ImageJ knows five image types:

- 8 bit grayscale image: can display 256 grayscales, a pixel is represented by a `byte` number.
- 8 bit color image: can display 256 colors that are specified in a lookup table (LUT), a pixel is represented by a `byte` number.
- 16 bit grayscale image: can display 65.536 grayscales, a pixel is represented by a `short` number.
- RGB color image: can display 256 values per channel, a pixel is represented by an `int.`number.
- 32 bit image: floating point grayscale image, a pixel is represented by a `float` number.

4.2 Images

An ImagePlus is an object that represents an image. It is based on an ImageProcessor, a class that holds the pixel array and does the actual work on the image. The type of the ImageProcessor used depends on the type of the image. The image types are represented by constants in ImagePlus:

`COLOR_256`
: A 8 bit color image with a look-up table.

`COLOR_RGB`
: A RGB color image.

`GRAY16`
: A 16 bit grayscale image.

`GRAY32`
: A 32 bit floating point grayscale image.

`GRAY8`
: A 8 bit grayscale image.

ImageJ displays images using a class called ImageWindow. It handles repainting, zooming, changing masks etc.

To construct an ImagePlus use one of the following constructors:

`ImagePlus()`

Default constructor, creates a new empty ImagePlus and does no initialization.

`ImagePlus(java.lang.String urlString)`

Constructs a new ImagePlus, loading the Image from the URL specified.

`ImagePlus(java.lang.String title, java.awt.Image img)`

Constructs a new ImagePlus based on a Java AWT image. The first argument is the title of the ImageWindow that displays the image.

`ImagePlus(java.lang.String title, ImageProcessor ip)`

Constructs a new ImagePlus that uses the specified ImageProcessor. The first argument is the title of the ImageWindow that displays the image.

`ImagePlus(java.lang.String title, ImageStack stack)`

Constructs a new ImagePlus from an ImageStack. The first argument is the title of the ImageWindow that displays the image.

The type of an ImagePlus can be retrieved using

```
int getType()
```

Similar methods exist for the image dimension, the title (= name of the ImageWindow that displays this image) and the URL the image was loaded from:

```
int getHeight()
int getWidth()
java.lang.String getTitle()
java.net.URL getURL()
```

The AWT image the ImagePlus is based on and the title can be set using

```
void setImage(java.awt.Image img)
void setTitle(java.lang.String title)
```

4.3 Processors

Each image is based on an "image processor". The type of the processor depends on the ImageType. You can get and set the image processor using these two methods of an ImagePlus:

`ImageProcessor getProcessor()`

Returns a reference to the current ImageProcessor.

`void setProcessor(java.lang.String title, ImageProcessor ip)`

Sets the image processor to the one specified.

When working with plugin filters you do not have to care about retrieving the processor from the `ImagePlus`, it is passed as argument to the `run` method.

`ImageProcessor` is an abstract class. Depending on the type of the image we use a subclass of ImageProcessor. There are five of them:

`ByteProcessor`

Used for 8 bit grayscale and color images. It has a subclass called `BinaryProcessor` for grayscale images that contain pixel values 0 and 255.

`ShortProcessor`

Used for 16 bit grayscale images.

`ColorProcessor`

Used for 32 bit integer images (RGB with 8 bit/channel).

`FloatProcessor`

Used for 32 bit floating point images.

4.4 Accessing Pixel Values

To work with the image we need access to it's pixels. We know how to get the image's ImageProcessor. Retrieving the pixel values can be done by using an ImageProcessor's

```
java.lang.Object getPixels()
```

method. It returns a reference to this image's pixel array. As the type of this array depends on the image type we need to cast this array to the appropriate type when we get it.

```
ColorProcessor myProcessor = myImage.getProcessor();
int[] pixels = (int[]) myProcessor.getPixels();
```

This example would work for an RGB image. As you have noticed we get back a one-dimensional array. It contains the image scanline by scanline. To convert a position in this array to a (x,y) coordinate in an image, we need at least the width of a scanline. The width and height of an ImageProcessor can be retrieved using these methods:

```
int getHeight()
int getWidth()
```

Now we have everything to iterate through the pixel array. As you have seen in the sample plugin this can be done using two nested loops.

Two cases need a bit more explanation: Reading pixels from `ByteProcessors` and from `ColorProcessors`.

Java's `byte` data type is *signed* and has values ranging from – 128 to 127, while we would expect a 8 bit grayscale image to have values from 0 to 255. If we cast a `byte` variable to another type we have to make sure that the sign bit is eliminated. This can be done using a binary AND:

```
int pix = 0xff & pixels[i];
...
pixels[i] = (byte) (pix & 0xff);
```

ColorProcessors return the pixel array as an `int[]`. The values of the three color components are packed into one `int`. They can be accessed as follows:

```
int red   = (int)(pixels[i] & 0xff0000)>>16;
int green = (int)(pixels[i] & 0x00ff00)>>8;
int blue  = (int)(pixels[i] & 0x0000ff);
...
pixels[i] = ((red & 0xff) << 16) + ((green & 0xff) << 8) + (blue & 0xff);
```

The pixel array you work on is just a reference to the ImageProcessor's pixel array. So any modifications effect the ImageProcessor immediately. However, if you want the ImageProcessor to use another (perhaps newly created) array, you can do this using

```
void setPixels(java.lang.Object pixels)
```

You do not always have to retrieve or set the whole pixel array. ImageProcessor offers some other methods for retrieving or setting pixel values:

`int getPixel(int x, int y)`

Returns the value of the specified pixel.

`void putPixel(int x, int y, int value)`

Sets the pixel at (`x`, `y`) to the specified value.

```
float getPixelValue(int x, int y)
```
Returns the value of the specified pixel.
```
void getColumn(int x, int y, int[] data, int length)
```
Returns the pixels down the column starting at (`x`, `y`).
```
void putColumn(int x, int y, int[] data, int length)
```
Inserts the pixels contained in `data` into a column starting at (`x`, `y`).
```
void getRow(int x, int y, int[] data, int length)
```
Returns the pixels along the horizontal line starting at (x,y).
```
void putRow(int x, int y, int[] data, int length)
```
Inserts the pixels contained in `data` into a horizontal line starting at (x,y).
```
double[] getLine(int x1, int y1, int x2, int y2)
```
Returns the pixels along the line (`x1`,`y1`)/(`x2`,`y2`).

The method
```
int[] getPixel(int x, int y)
```
of ImagePlus returns the pixel value at (x,y) as a 4 element array.

All these methods should be used if you intend to modify just a few pixels. If you want to modify large parts of the image it is faster to work with the pixel array.

4.5 Regions of Interest

A plugin filter does not always have to work on the whole image. ImageJ supports regions of interest (ROI), rectangular, circular, polygonal or freeform selections of regions of the image.

The bounding rectangle of the current ROI can be retrieved from the ImageProcessor using
```
java.awt.Rectangle getRoi()
```
This makes it possible to just handle the pixels that are inside this rectangle. It is also possible to set a processors ROI:
```
void setRoi(int x, int y, int rwidth, int rheight)
```
This sets the ROI to the rectangle starting at (`x`,`y`) with specified width and height.

More methods for working with ROIs can be found in ImagePlus. Remember that a plugin filter's `run` method receives an ImageProcessor as argument, but you can access the ImagePlus in the `setup` method.
```
void setRoi(int x, int y, int width, int height)
```
Creates a rectangular selection starting at (x,y) with specified width and height.
```
void setRoi(java.awt.Rectangle r)
```
Creates a rectangular selection.
```
void setRoi(Roi roi)
```
Creates a selection based on the specified ROI object.
```
Roi getRoi()
```
Returns a ROI object representing the current selection.

4.6 Creating New Images

In many cases it will make sense that a plugin does not modify the original image, but creates a new image that contains the modifications.

ImagePlus' method

`ImagePlus createImagePlus()`

returns a new ImagePlus with this ImagePlus' attributes, but no image. A similar function is provided by ImageProcessor's

`ImageProcessor createProcessor(int width, int height)`

which returns a new, blank processor with specified width and height which can be used to create a new ImagePlus using the constructor

`ImagePlus(java.lang.String title, ImageProcessor ip)`

The class `NewImage` offers some useful static methods for creating a new `ImagePlus` of a certain type.

```
static ImagePlus createByteImage(java.lang.String title,
                                 int width, int height,
                                 int slices, int fill)
```

Creates a new 8 bit grayscale or color image with the specified title, width and height and number of slices. `fill` is one of the constants listed below that determine how the image is initially filled.

```
static ImagePlus createFloatImage(java.lang.String title,
                                  int width, int height,
                                  int slices, int fill)
```

Creates a new 32 bit floating point image with the specified title, width and height and number of slices. `fill` is one of the constants listed below that determine how the image is initially filled.

```
static ImagePlus createRGBImage(java.lang.String title,
                                int width, int height,
                                int slices, int fill)
```

Creates a new RGB image with the specified title, width and height and number of slices. `fill` is one of the constants listed below that determine how the image is initially filled.

```
static ImagePlus createShortImage(java.lang.String title,
                                  int width, int height,
                                  int slices, int fill)
```

Creates a new 16 bit grayscale image with the specified title, width and height and number of slices. `fill` is one of the constants listed below that determine how the image is initially filled.

These are the possible values for the `fill` argument defined in class `NewImage:`

`FILL_BLACK`

Fills the image with black color.

`FILL_WHITE`

Fills the image with white color

`FILL_RAMP`

Fills the image with a horizontal grayscale ramp.

There are two methods to copy pixel values between different ImageProcessors:

`void copyBits(ImageProcessor ip, int xloc, int yloc, int mode)`

Copies the image represented by `ip` to `xloc`, `yloc` using the specified blitting mode. This is one of the following constants defined in the interface `Blitter`:

`ADD`.................................. destination = destination+source

`AND` .... destination = destination AND source
`AVERAGE`......................... destination = (destination+source)/2
`COPY`................................ destination = source
`COPY_INVERTED` .... destination = 255-source
`COPY_TRANSPARENT`... White pixels are assumed as transparent.
`DIFFERENCE` .... destination = |destination-source|
`DIVIDE` .... destination = destination/source
`MAX` .... destination = maximum(destination,source)
`MIN` .... destination = minimum(destination,source)
`MULTIPLY`...................... destination = destination*source
`OR`..................................... destination = destination OR source
`SUBTRACT`...................... destination = destination-source
`XOR`.................................. destination = destination XOR source

```
void insert(ImageProcessor ip, int xloc, int yloc)
```

Inserts the image contained in `ip` at (`xloc`, `yloc`).

If you do not need a new ImagePlus for use in ImageJ but a Java AWT image you can retrieve it from the image processor using

```
java.awt.Image createImage()
```

The same function is provided by ImagePlus'

```
java.awt.Image getImage()
```

4.7 Displaying Images

Now that we can modify images we need to know how the changes can be made visible. ImageJ uses a class called ImageWindow to display ImagePlus images. ImagePlus contains everything that is necessary for updating or showing newly created images.

```
void draw()
```

Displays this image.

```
void draw(int x, int y, int width, int height)
```

Draws image and the ROI outline using a clipping rectangle.

```
void updateAndDraw()
```

Updates this image from the pixel data in its associated ImageProcessor, then displays it.

```
void updateAndRepaintWindow()
```

Calls `updateAndDraw` to update from the pixel data and draw the image, and also repaints the image window to force the information displayed above the image (dimension, type, size) to be updated.

```
void show()
```

Opens a window to display this image and clears the status bar.

```
void show(java.lang.String statusMessage)
```

Opens a window to display this image and displays `statusMessage` in the status bar.

```
void hide()
```

Closes the window, if any, that is displaying this image.

4.8 ColorInverter PlugIn (Example)

With the knowledge of the previous sections we can write our first own plugin. We will modify the Inverter plugin so that it handles RGB images. It will invert the colors of the pixels of the original image's ROI and display the result in a new window.

As mentioned before, we start from the existing plugin Inverter_. First of all we modify the class name.

```
import ij.plugin.filter.PlugInFilter;
import ij.process.*;
import java.awt.*;
public class ColorInverter_ implements PlugInFilter {
  ...
```

Don't forget to rename the file to `ColorInverter_.java`, otherwise you won't be able to compile it.

We want to handle RGB files, we do not want to apply it to stacks, we want to support non-rectangular ROIs and we because we display the results in a new image we do not modify the original, so we change the capabilities returned by the setup method to `DOES_RGB+SUPPORTS_MASKING+NO_CHANGES`.

```
public int setup(String arg, ImagePlus imp) {
  if (arg.equals("about")) {
    showAbout(); return DONE;
  }
  return DOES_RGB+SUPPORTS_MASKING+NO_CHANGES;
}
```

The run method will do the actual work.

```
public void run(ImageProcessor ip) {
```

First we save the dimension and the ROI of the original image to local variables.

```
int w = ip.getWidth();
int h = ip.getHeight();
Rectangle roi = ip.getRoi();
```

We want to have the result written to a new image, so we create a new RGB image of the same size, with one slice and initially black and get the new image's processor.

```
ImagePlus inverted = NewImage.createRGBImage ("Inverted image", w, h,
                                              1, NewImage.FILL_BLACK);
ImageProcessor inv_ip = inverted.getProcessor();
```

Then we copy the image from the original ImageProcessor to (0,0) in the new image, using `COPY` blitting mode (this mode just overwrites the pixels in the destination processor). We then get the pixel array of the new image (which is of course identical to the old one). It's a RGB image, so we get an `int` array.

```
inv_ip.copyBits(ip,0,0,Blitter.COPY);
int[] pixels = (int[]) inv_ip.getPixels();
```

We now go through the bounding rectangle of the ROI with two nested loops. The outer one runs through the lines in the ROI, the inner one through the columns in each line. The offset in the one-dimensional array is the start of the current line (= width of the image × number of scanlines).

```
for (int i=roi.y; i<roi.y+roi.height; i++) {
  int offset =i*w;
  for (int j=roi.x; j<roi.x+roi.width; j++) {
```

In the inner loop we calculate the position of the current pixel in the one-dimensional array (we save it in a variable because we need it twice). We then get the value of the current pixel. Note that we can access the pixel array of the new image, as it contains a copy of the old one.

```
            int pos = offset+j;
            int c = pixels[pos];
```

We extract the three color components as described above.

```
            int r = (c&0xff0000)>>16;
            int g = (c&0x00ff00)>>8;
            int b = (c&0x0000ff);
```

We invert each component by subtracting it's value from 255. Then we pack the modified color components into an integer again.

```
            r=255-r;
            g=255-g;
            b=255-b;
            pixels[pos] = ((r & 0xff) << 16) +
                          ((g & 0xff) << 8) +
                           (b & 0xff);
          }
        }
```

We have now done all necessary modifications to the pixel array. Our image is still not visible, so we call `show` to open an ImageWindow that displays it. Then we call `updateAndDraw` to force the pixel array to be read and the image to be updated.

```
        inverted.show();
        inverted.updateAndDraw();
      }
    }
```

4.9 Stacks

ImageJ supports expandable arrays of images called image stacks, that consist of images (slices) of the same size. In a plugin filter you can access the currently open stack by retrieving it from the current ImagePlus using

```
ImageStack getStack()
```

ImagePlus also offers a method for creating a new stack:

```
ImageStack createEmptyStack()
```

Returns an empty image stack that has the same width, height and color table as this image.

Alternatively you can create an ImageStack using one of these constructors:

```
ImageStack(int width, int height)
```

Creates a new, empty image stack with specified height and width.

```
ImageStack(int width, int height, java.awt.image.ColorModel cm)
```

Creates a new, empty image stack with specified height, width and color model.

To set the newly created stack as the stack of the current image use

```
void setStack(java.lang.String title, ImageStack stack)
```

Sets the specified image stack with the specified title as stack of an ImagePlus.

The number of slices of a stack can be retrieved using the methods

```
int getSize()
```

of class ImageStack or

```
int getStackSize()
```

of class ImagePlus.

The currently displayed slice of an ImagePlus can be retrieved and set using

```
int getCurrentSlice()
void setSlice(int index)
```

A stack offers several methods for retrieving and setting its properties:

`int getHeight()`

Returns the height of the stack.

`int getWidth()`

Returns the width of the stack.

`java.lang.Object getPixels(int n)`

Returns the pixel array for the specified slice, where `n` is a number from 1 to the number of slices. See also section 4.4.

`void setPixels(java.lang.Object pixels, int n)`

Assigns a pixel array to the specified slice, where `n` is a number from 1 to the number of slices. See also section 4.4.

`ImageProcessor getProcessor(int n)`

Returns an ImageProcessor for the specified slice, where `n` is a number from 1 to the number of slices. See also section 4.3.

`java.lang.String getSliceLabel(int n)`

Returns the label of the specified slice, where `n` is a number from 1 to the number of slices.

`void setSliceLabel(java.lang.String label, int n)`

Sets the label of the specified slice, where `n` is a number from 1 to the number of slices.

`java.awt.Rectangle getRoi()`

Returns the bounding rectangle of the stack's ROI. For more information on ROIs see section 4.5.

`void setRoi(java.awt.Rectangle roi)`

Sets the stack's ROI to the specified rectangle. For more information on ROIs see section 4.5.

Slices can be added to and removed from the stack using these methods:

`void addSlice(java.lang.String sliceLabel, ImageProcessor ip)`

Adds the image represented by `ip` to the end of the stack.

`void addSlice(java.lang.String sliceLabel, ImageProcessor ip, int n)`

Adds the image represented by `ip` to the stack following slice `n`.

`void addSlice(java.lang.String sliceLabel, java.lang.Object pixels)`

Adds an image represented by its pixel array to the end of the stack.

`void deleteLastSlice()`

Deletes the last slice in the stack.

`void deleteSlice(int n)`

Deletes the specified slice, where `n` is in the range 1 .. number of slices.

4.10 StackAverage PlugIn (Example)

This example shows how to handle stacks. It calculates the average values of pixels located at the same position in each slice of the stack and adds an image showing the average values to the end of the stack.

First of all, we import the necessary packages. We want to work on the current stack so we need to implement `PlugInFilter`.

```
import ij.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;
public class StackAverage_ implements PlugInFilter {
  ...
```

We define the stack as instance variable because we will retrieve it in `setup` and use it in `run`.

```
    protected ImageStack stack;
```

In this method we get the stack from the current image and return the plugin's capabilities – in this case we indicate that it handles 8 bit grayscale images and requires a stack as input.

```
    public int setup(String arg, ImagePlus imp) {
      stack = imp.getStack();
      return DOES_8G+STACK_REQUIRED;
    }
```

In the run method we declare a `byte` array that will hold the pixels of the current slice. Then we get width and height of the stack and calculate the length of the pixel array of each slice as the product of width and height. `sum` is the array to hold the summed pixel values.

```
    public void run(ImageProcessor ip) {
      byte[] pixels;
      int dimension = stack.getWidth()*stack.getHeight();
      sum = new long[dimension];
```

In the outer loop we iterate through the slices of the stack and get the pixel array from each slice. In the inner loop we go through the pixel array of the current slice and add the pixel value to the corresponding pixel in the `sum` array.

```
      for (int i=1;i<=stack.getSize();i++) {
        pixels = (byte[]) stack.getPixels(i);
        for (int j=0;j<dimension;j++) {
          sum[j]+=0xff & pixels[j];
        }
      }
```

We have now gone through the whole stack. The image containing the averages will be a 8 bit grayscale image again, so we create a `byte` array for it. Then we iterate through the pixels in the `sum` array and divide each of them through the number of slices to get pixel values in the range 0..255.

```
      byte[] average = new byte[dimension];
      for (int j=0;j<dimension;j++) {
        average[j] = (byte) ((sum[j]/stack.getSize()) & 0xff);
      }
```

Finally we add a new slice to the stack. It is called "Average" and represented by the pixel array that contains the average values.

```
      stack.addSlice("Average",average);
    }
```

4.11 Additional Reference

This reference is thought as a supplement to the concepts presented in this section. It is not complete – it just covers what you will normally need for writing plugins. For a complete reference see the API documentation and/or the source code.

4.11.1 ImagePlus

ImagePlus and ImageWindow

`void setWindow(ImageWindow win)`

Sets the the window that displays the imgage.

`ImageWindow getWindow()`

Gets the window that is used to display the image.

`void mouseMoved(int x, int y)`

Displays the cursor coordinates and pixel value in the status bar.

Multithreading

`boolean lock()`

Locks the image so that it cannot be accessed by another thread.

`boolean lockSilently()`

Similar to lock, but doesn't beep and display an error message if the attempt to lock the image fails.

`void unlock()`

Unlocks the image.

Lookup Tables

`LookUpTable createLut()`

Creates a LookUpTable based on the image.

4.11.2 ImageProcessor

Geometric transforms

`void flipHorizontal()`

Flips the image horizontally.

`void flipVertical()`

Flips the image vertically.

`void rotate(double angle)`

Rotates the image `angle` degrees clockwise.

`void scale(double xScale, double yScale)`

Scales the image by the specified factors.

`ImageProcessor crop()`

Crops the image to the bounding rectangle of the current ROI. Returns a new image processor that represents the cropped image.

`ImageProcessor resize(int dstWidth, int dstHeight)`

Resizes the image to the specified destination size. Returns a new image processor that represents the resized image.

`ImageProcessor rotateLeft()`

Rotates the image 90 degrees counter-clockwise. Returns a new image processor that represents the rotated image.

`ImageProcessor rotateRight()`

Rotates the image 90 degrees clockwise. Returns a new image processor that represents the rotated image.

`void setInterpolate(boolean interpolate)`

Setting interpolate true causes `scale()`, `resize()` and `rotate()` to do bilinear interpolation.

Filters

`void convolve3x3(int[] kernel)`

Convolves the image with the specified 3x3 convolution matrix. The following methods are based on `convolve`:

`void sharpen()`

Sharpens the image using a 3x3 convolution kernel.

`void smooth()`

Replaces each pixel with the 3x3 neighborhood mean.

`void filter(int type)`

A 3x3 filter operation, the argument defines the filter type. The following methods are based on `filter`:

`void dilate()`

Dilates the image using a 3x3 minimum filter.

`void erode()`

Erodes the image using a 3x3 maximum filter.

`void findEdges()`

Finds edges using a Sobel operator.

`void medianFilter()`

A 3x3 median filter.

`void gamma(double value)`

A gamma correction.

`void invert()`

Inverts an image.

`void add(int value)`

Adds the argument to each pixel value.

`void multiply(double value)`

Multiplies each pixel value with the argument.

`void and(int value)`

Binary AND of each pixel value with the argument.

`void or(int value)`

Binary OR of each pixel value with the argument.

`void xor(int value)`

Binary exclusive OR of each pixel value with the argument.

`void log()`

Calculates pixel values on a logarithmic scale.

`void noise(double range)`

Adds random noise (random numbers within `range`) to the image.

Drawing

`void setColor(java.awt.Color color)`

Sets the foreground color. This will set the default fill/draw value to the pixel value that represents this color.

`void setValue(double value)`

Sets the default fill/draw value.

`void setLineWidth(int width)`

Sets the line width.

`void moveTo(int x, int y)`

Sets the current drawing location to (`x`,`y`).

`void lineTo(int x2, int y2)`

Draws a line from the current drawing location to (`x2`,`y2`).

`void drawPixel(int x, int y)`

Sets the pixel at (`x`,`y`) to the current drawing color.

`void drawDot(int xcenter, int ycenter)`

Draws a dot using the current line width and color.

`void drawDot2(int x, int y)`

Draws 2x2 dot in the current color.

`void fill()`

Fills the current rectangular ROI with the current drawing color.

`void fill(int[] mask)`

Fills pixels that are within the current ROI and part of the mask (i.e. pixels that have value 0 = black in the mask array).

`void drawString(java.lang.String s)`

Draws a string at the current location with the current color.

`int getStringWidth(java.lang.String s)`

Returns the width in pixels of the specified string.

Colors

`int getBestIndex(java.awt.Color c)`

Returns the LUT index that's the best match for this color.

`java.awt.image.ColorModel getColorModel()`

Returns this processor’s color model.

`void invertLut()`

Inverts the values in the lookup table.

Minimum, Maximum and Threshold

`double getMin()`

Returns the smallest displayed pixel value.

`double getMax()`

Returns the largest displayed pixel value.

`void setMinAndMax(double min, double max)`

Maps the pixels in this image from min…max to 0…255.

`void resetMinAndMax()`

For short and float images, recalculates the min and max image values needed to correctly display the image.

`void autoThreshold()`

Calculates auto threshold of an image and applies it.

`double getMinThreshold()`

Returns the minimum threshold.

`double getMaxThreshold()`

Returns the maximum threshold.

`void setThreshold(double minThreshold, double maxThreshold, int lutUpdate)`

Sets the minimum and maximum threshold levels.

Histograms

`int[] getHistogram()`

Returns the histogram of the image. This method will return a luminosity histogram for RGB images and `null` for floating point images.

`int getHistogramSize()`

The size of the histogram is 256 for 8 bit and RGB images and max–min+1 for 16 bit integer images.

Snapshots (Undo)

`void snapshot()`

Saves the current state of the processor as snapshot.

`java.lang.Object getPixelsCopy()`

Returns a reference to this image's snapshot (undo) array, this is the pixel array before the last modification.

`void reset()`

Resets the processor to the state saved in the snapshot.

`void reset(int[] mask)`

Resets the processor to the state saved in the snapshot, excluding pixels that are part of `mask`.

4.11.3 ImageStack

Accessing Images

`java.lang.Object[] getImageArray()`

Returns the stack as an array of ImagePlus objects.

Color

`boolean isHSB()`

Returns true if this is a 3-slice HSB stack.

`boolean isRGB()`

Returns true if this is a 3-slice RGB stack.

`java.awt.image.ColorModel getColorModel()`

Returns this stack's color model.

`void setColorModel(java.awt.image.ColorModel cm)`

Assigns a new color model to this stack.

5 Using ImageJ's Utility Methods

The ImageJ API contains a class called `IJ` that contains some very useful static methods.

5.1 (Error) Messages

It is often necessary that a plugin displays a message – be it an error message or any other information. In the first case you will use

`static void error(java.lang.String msg)`

which displays a message in a dialog box titled "Error", in the second case

`static void showMessage(java.lang.String msg)`

which displays a message in a dialog box titled "Message". You can also specify the title of the message box using

`static void showMessage(java.lang.String title, java.lang.String msg)`

All these methods display messages that the user has to accept. If you want to let the user chose whether to cancel the plugin or to let it continue use

```
static boolean showMessageWithCancel(java.lang.String title,
                                     java.lang.String msg)
```

This method returns false if the user clicked cancel, true otherwise.

There are also some predefined messages:

`static void noImage()`

Displays a "no images are open" dialog box.

`static void outOfMemory(java.lang.String name)`

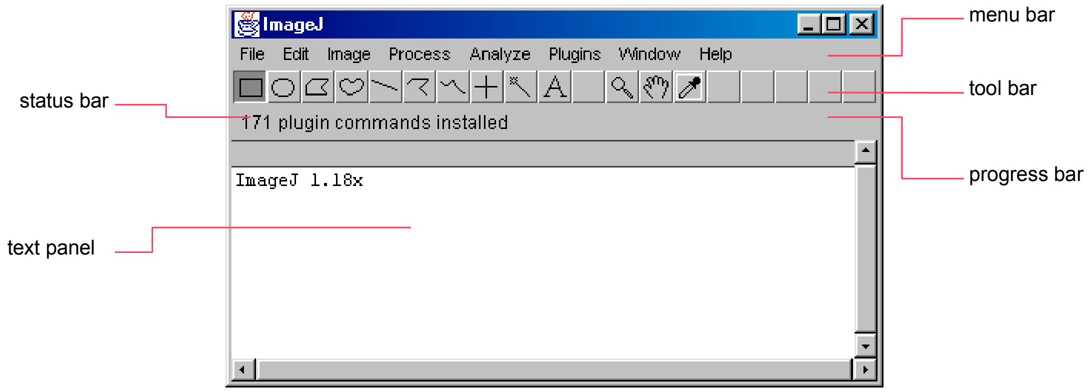
Displays an "out of memory" message in the ImageJ window.

`static boolean versionLessThan(java.lang.String version)`

Displays an error message and returns false if the ImageJ version is less than the one specified.

5.2 ImageJ Window, Status Bar and Progress Bar

The ImageJ main window consists of the following components:



Displaying Text

To display a line of text in the text panel (as "ImageJ 1.18x" in the screenshot) use

```
static void write(java.lang.String s)
```

It is possible to use the text panel as a table (e.g. for displaying statistics, measurements, etc.). In that case ImageJ lets you set the headings of the columns using

```
static void setColumnHeadings(java.lang.String headings)
```

Note that this method clears the entire text panel.

You will often want to displays numbers, which you can format for output using

```
static java.lang.String d2s(double n)
```

Converts a number to a formatted string using two digits to the right of the decimal point.

```
static java.lang.String d2s(double n, int precision)
```

Converts a number to a rounded formatted string.

Status Bar

Text can also be displayed in the status bar above the text panel using the method

```
static void showStatus(java.lang.String s)
```

It can be useful to display the time that was needed for an operation.

```
static void showTime(ImagePlus imp, long start, java.lang.String str)
```

will display the string argument you specify, followed by the time elapsed since the specified start value and the rate of processed pixels per second.

Progress Bar

The progress of the current operation can be visualized using ImageJ's progress bar.

```
static void showProgress(double progress)
```

updates the position of the progress bar to the specified value (in the range from 0.0 to 1.0).

5.3 User input

Often user input (e.g. a parameter) is required in a plugin. ImageJ offers two simple methods for that purpose.

```
static double getNumber(java.lang.String prompt, double defaultNumber)
```

Allows the user to enter a number in a dialog box.

```
static java.lang.String getString(java.lang.String prompt,
                                  java.lang.String defaultString)
```

Allows the user to enter a string in a dialog box.

A way to build more sophisticated dialogs is presented in section 6.2.

5.4 Calling Menu Commands

You can access all menu commands from a plugin. There are two different methods:

```
static void doCommand(java.lang.String command)
```

Starts executing a menu command in a separate thread and returns immediately. Executing the command in a separate thread means that the program will not wait until the command is executed, it will immediately proceed. This has the advantage that the program is not blocked while the command is running.

```
static void run(java.lang.String command)
```

Runs a menu command in the current thread, the program is will continue after the command has finished.

5.5 Calling Other PlugIns

Like menu commands you can also run other plugins.

```
static java.lang.Object runPlugIn(java.lang.String className,
                                  java.lang.String arg)
```

Runs the plugin specified by its class name and initializes it with the specified argument.

5.6 MessageTest PlugIn (Example)

We will now look at a plugin that uses some of the utility methods presented in this chapter. This time, we do not need an image, so we implement the interface `PlugIn`. We also have to import the package `ij` as we need `IJ` from there.

```
import ij.*;
import ij.plugin.PlugIn;
public class Message_Test implements PlugIn {
```

All we have to implement is the run method. We do not need the argument, so we ignore it. First of all we display a string in the status bar that informs the user that the plugin was started. Then we set the progress bar to 0% and show an error message.

```
public void run(String arg) {
     IJ.showStatus("Plugin Message Test started.");
     IJ.showProgress(0.0);
     IJ.error("I need user input!");
```

We want the user to input a string and set the progress bar to 50% after that. Then we write a message into the main window saying that we were going to start the sample plugin `RedAndBlue` (this is one of the plugins that come with ImageJ and displays a new image with a red/blue gradient) and run the plugin. Finally we set the progress bar to 100% and show a custom message box.

```
     String name = IJ.getString("Please enter your name: ",
          "I.J. User");
     IJ.showProgress(0.5);
```

```
        IJ.write("Starting sample plugin RedAndBlue ... ");
        IJ.runPlugIn("RedAndBlue_","");
        IJ.showProgress(1.0);
        IJ.showMessage("Finished.",name+", thank you for running this
            plugin");
    }
}
```

5.7 More utilities

Keyboard & Sound

`static void beep()`

Emits an audio beep.

`static boolean altKeyDown()`

Returns true if the alt key is down.

`static boolean spaceBarDown()`

Returns true if the space bar is down.

Accessing GUI Elements

`static ImageJ getInstance()`

Returns a reference to the "ImageJ" frame.

`static java.applet.Applet getApplet()`

Returns the applet that created this ImageJ or `null` if running as an application.

`static TextPanel getTextPanel()`

Returns a reference to ImageJ's text panel.

Misc

`static boolean isMacintosh()`

Returns true if this machine is a Macintosh.

`static void wait(int msecs)`

Delays `msecs` milliseconds.

`static java.lang.String freeMemory()`

Returns the amount of free memory in KB as string.

6 Windows

By default, plugins work with ImagePlus objects displayed in ImageWindows. They can output information to the ImageJ window but they cannot control a window. Sometimes this can be necessary, especially for getting user input.

6.1 PlugInFrame

A PlugInFrame is a subclass of an AWT frame that implements the `PlugIn` interface. Your plugin will be implemented as a subclass of PlugInFrame.

There is one constructor for a PlugInFrame. It receives the title of the window as argument:

```
PlugInFrame(java.lang.String title)
```

As this class is a plugin, the method

```
void run(java.lang.String arg)
```

declared in the `PlugIn` interface is implemented and can be overwritten by your plugin's `run` method.

Of course all methods declared in `java.awt.Frame` and its superclasses can be overwritten. For details consult the AWT API documentation.

6.2 GenericDialog

In section 5.3 we saw a very simple method of getting user input. If you need more user input than just one string or number, GenericDialog helps you build a modal (that means that the programs only proceeds after the user has answered the dialog) AWT dialog. The GenericDialog can be built on the fly and you don't have to care about event handling.

There are two constructors:

```
GenericDialog(java.lang.String title)
```

Creates a new GenericDialog with the specified title.

```
GenericDialog(java.lang.String title, java.awt.Frame parent)
```

Creates a new GenericDialog using the specified title and parent frame. The ImageJ frame can be retrieved using `IJ.getInstance()`.

The dialog can be displayed using

```
void showDialog()
```

Adding controls

GenericDialog offers several methods for adding standard controls to the dialog:

```
void addCheckbox(java.lang.String label, boolean defaultValue)
```

Adds a checkbox with the specified label and default value.

```
void addChoice(java.lang.String label,
               java.lang.String[] items,
               java.lang.String defaultItem)
```

Adds a drop down list (popup menu) with the specified label, items and default value.

```
void addMessage(java.lang.String text)
```

Adds a message consisting of one or more lines of text.

```
void addNumericField(java.lang.String label,
                     double defaultValue, int digits)
```

Adds a numeric field with the specified label, default value and number of digits.

```
void addStringField(java.lang.String label,
                    java.lang.String defaultText)
```

Adds a 8 column text field with the specified label and default value.

```
void addStringField(java.lang.String label,
                    java.lang.String defaultText, int columns)
```

Adds a text field with the specified label, default value and number of columns.

```
void addTextAreas(java.lang.String text1,
                  java.lang.String text2,
                  int rows, int columns)
```

Adds one or two text areas (side by side) with the specified initial contents and number of rows and columns. If `text2` is `null`, the second text area will not be displayed.

Getting Values From Controls

After the user has closed the dialog window, you can access the values of the controls with the methods listed here. There is one method for each type of control. If the dialog contains more than one control of the same type, each call of the method will return the value of the next control of this type in the sequence they were added to the dialog.

```
boolean getNextBoolean()
```

Returns the state of the next checkbox.

```
java.lang.String getNextChoice()
```

Returns the selected item in the next drop down list (popup menu).

```
int getNextChoiceIndex()
```

Returns the index of the selected item in the next drop down list (popup menu).

```
double getNextNumber()
```

Returns the contents of the next numeric field.

```
java.lang.String getNextString()
```

Returns the contents of the next text field.

```
java.lang.String getNextText()
```

Returns the contents of the next text area.

The method

```
boolean wasCanceled()
```

returns true, if the user closed the dialog using the cancel button, and false, if the user clicked the OK button.

If the dialog contains numeric fields, use

```
boolean invalidNumber()
```

to check if the values in the numeric fields are valid numbers. This method returns true if at least one numeric filed does not contain a valid number.

GenericDialog extends AWT Dialog, so you can use any method of `java.awt.Dialog` or one of ist superclasses. For more information consult the AWT documentation.

6.3 FrameDemo PlugIn (Example)

This demo shows the usage of GenericDialog and PlugInFrame. It displays a dialog that lets the user specify the width and height of the PlugInFrame that will be displayed after closing the dialog.

We import the `ij` and `ij.process` package, the `ij.gui` package, where GenericDialog is located and the classes PlugInFrame and AWT Label.

```
import ij.*;
import ij.process.*;
import ij.gui.*;
```

```
import ij.plugin.frame.PlugInFrame;
import java.awt.Label;
```

Our plugin is a subclass of PlugInFrame which implements the `PlugIn` interface, so we don't have to implement an interface here.

```
public class FrameDemo_ extends PlugInFrame {
```

We overwrite the default constructor of the new class. If we wouldn't do that, the superclass' default constructor `PlugInFrame()` would be called, which does not exist. So we have to call the superclass' constructor and specify a title for the new frame.

```
    public FrameDemo_() {
        super("FrameDemo");
    }
```

In the `run` method we create a GenericDialog with the title "FrameDemo settings". Then we add two 3 digit numeric fields with a default value of 200.

```
    public void run(String arg) {
        GenericDialog gd = new GenericDialog("FrameDemo settings");
        gd.addNumericField("Frame width:",200.0,3);
        gd.addNumericField("Frame height:",200.0,3);
```

We show the dialog. As it is modal, the program is stopped until the user closes the dialog. If the user clicks "Cancel" we display an error message and leave the `run` method.

```
        gd.showDialog();
        if (gd.wasCanceled()) {
            IJ.error("PlugIn canceled!");
            return;
        }
```

Here we get the values of the numeric fields with two calls of `getNextNumber()`. We set the size of the FrameDemo window to these values and add a centered AWT Label with the text "PlugInFrame demo". Finally we show the frame.

```
        this.setSize((int) gd.getNextNumber(),(int) gd.getNextNumber());
        this.add(new Label("PlugInFrame demo",Label.CENTER));
        this.show();
    }
}
```

7 Troubleshooting

8 Frequently Asked Questions

9 Additional Resources

9.1 API Documentation, Source Code

The ImageJ API documentation is available online at
`http://rsb.info.nih.gov/ij/docs/api/index.html`.

API documentation and source code are available for download at
`http://rsb.info.nih.gov/ij/download.html`.

9.2 PlugIns Page

Many ImageJ plugins (with source code) are available at
`http://rsb.info.nih.gov/ij/plugins/index.html`.

9.3 ImageJ Mailing List

For questions concerning ImageJ that are not answered by the documentation consult the ImageJ mailing list.

A complete archive can be found at `http://list.nih.gov/archives/imagej.html`

For information about subscribing see `http://rsb.info.nih.gov/ij/list.html`

9.4 Java Resources

Online Resources

Java API documentation and many tutorials are available from Sun Microsystems at `http://java.sun.com/` under "Docs & Training".

Books

Java in a Nutshell: A Desktop Quick Reference (Java Series)
by David Flanagan
648 pages 3rd edition (November 1999)
O'Reilly & Associates
ISBN: 1565924878

Java Examples in a Nutshell
by David Flanagan
500 pages 2nd edition (September 2000)
O'Reilly & Associates
ISBN: 0596000391
The first third of this book is interesting for someone who wants to get into Java programming, the other chapters cover more advanced topics.

The Sun Java Series
Detailed information can be found at http://java.sun.com/docs/books/